# TCP/IP

陈昱先最爱小胖子!!!!!!!!!!!!!

## 网络七层

应用层        http协议:用于如何封装数据

表示层

会话层

传输层        tcp/udp协议:传输控制协议    解决数据在网络中的传输

网络层        ip协议:                解决数据在网络中的传输

数据链路层

物理层

## tcp三次握手

第一次握手:    客户端向服务端发送请求报文,并进入syn\_send状态

第二次握手:    服务器收到请求报文,若同意连接,会发送一个应答,服务器进入syn\_recv状态

第三次握手:    客户端收到服务器同意应答,向服务器发送一个确认报文,连接建立完成

为什么是三次不是两次?

    若只是两次握手,客户端没有什么影响,仍然是获得服务端应答才开始传输数据

    但是,服务端在接收到连接请求之后就进入传输状态,此时若是网络拥塞,客户端发送的连接请求迟迟不能到服务端,客户端便会超时重发,如果服务端收到这条消息,便建立连接,开始通信,结束后释放.但是如果此时失效的连接到达服务端,由于两次握手,服务端便会进入连接状态,等待发送数据或者主动发送数据,但此时客户端早已关闭,服务端会一直等下去,这样会浪费服务端资源

为什么不是四次?

    三次握手是保证双方能够收发信息的最低值,无论多少次握手都不能确保信道的可靠,单通过三次可以至少确认信道可用,再往上加握手次数也只是加大信道可用的可信程度

## tcp四次挥手

tcp的释放需要四步

前两次挥手用于断开一个方向的连接,后两次用于断开另一个方向

第一次挥手:    若A认为数据发送完成,他需要向B发送一个连接释放的请求

第二次挥手:    B收到连接释放请求后,会通知响应程序,告诉它A向B这个方向的连接已经释放,此时B进入close\_wait状态,并向A发送释放连接的应答    A收到该应答,进入fin\_wait\_2状态,等待B发送释放请求

    第二次挥手完成后,A到B方向的连接已经释放,B不会再接收数据,A也不会再发送数据,但是B到A方向的连接仍然存在,B可以继续向A发送数据

第三次挥手:    当B向A发送完所有数据后,向A发送连接释放请求,B进入last\_ack状态

第四次挥手:    A收到释放请求后,向B发送确认应答,此时A进入time\_wait状态,持续2sml,进入closed状态

    为什么A要先进入time\_wait状态,等待2msl再进入closed状态?

         客户端最后一次向服务端回传ack时,有可能因为网络情况服务端收不到,服务端会再次发送fin包,此时若客户端已经关闭,则服务端无法收到ack包,无法关闭,因此客户端需要等待2msl

         其中msl指的是数据包在网络中的生存时间,数据包在网络中有生存时间,超时未到达就会被丢弃 这个时间是1msl time\_wait需要2msl才进入closed,若2msl没有收到fin,说明服务端已关闭

## tcp与udp的区别

    tcp面向连接     在发送数据之前需要建立连接(三次握手)     udp是无连接的

    tcp提供可靠,有序的服务,且按序到达     udp尽最大努力交付,但是不保证交付可靠

    udp有较好的实时性,效率比tcp高     适用于高速传输和实时性有较高的通信或广播

    每一条tcp都是点对点     udp可以一对多     多对多

    tcp对系统资源要求较多,首部开销大     udp较少

## tcp报文

    源端口     目的端口     序号     确认序号     窗口:告知发送端接收端的缓存大小     数据

## 超时重传

         在发送某一个数据之后开启一个计时器,在一定时间内没有获得ack报文,则重新发送,知道成功或者最大重发次数,重传超时时间RTO     RTT连接往返的时间

## 滑动窗口

         提供tcp的可靠性     提供TCP的流控特性

         tcp的滑动窗口的可靠性是建立在确认重传的基础上的

         发送端窗口只有在接收端返回ack确认后,才会移动滑动窗口的左边界

         接收端可以根据自己的状况告知发送端窗口大小,控制发送端的收发,进行流量控制

## 拥塞控制

         拥塞控制就是防止过多的数据注入到网络中  发送方会维持一个叫做拥塞窗口cwnd的状态变量,拥塞窗口大小取决于网络的拥塞程度 ,发送窗口小于等于拥塞窗口

         拥塞控制是一个全局性过程     流量控制是点对点通信量的控制

         TCP拥塞控制有四个核心算法:慢开始     拥塞避免     快重传     快恢复

**拥塞窗口:** 大小取决于网络的拥塞程度,动态的变化

### 慢开始

不要一开始就发送大量的数据,先探测一下网络的拥塞程度,从小到大逐渐增加拥塞窗口的大小

             一次传输轮之后拥塞窗口就加倍,但是为了防止cwnd增长过大引起网络拥塞,会设置一个门限ssthresh

            当cwnd<ssthresh时  使用慢开始

                  cwnd>ssthresh     拥塞控制

                   cwnd=ssthresh    随机

### 拥塞避免

使得拥塞窗口缓慢增长,每经过一个往返时间就把拥塞窗口+1,实现线性增长(而慢开始时指数增长)

             无论是慢开始阶段还是拥塞避免阶段,只要发现网络出现拥塞现象,(没有收到确认包),就把慢开始门限ssthresh设为出现拥塞时发送窗口的一半,然后把拥塞窗口设为1,执行慢开始

### 快重传

接收方在接收到一个失序的报文段之后立刻发送重复确认,而不是等待自己返回数据时捎带确认

         发送方只要连续收到三个重复确认就应当立即重传对方尚未收到的报文段,而不必等待设置的重传计时器时间到来

         即发送方发送m1,m2,发送m3时丢失,  接收方回复三次重复请求,发送方继续发送m3

### 快恢复

1.当发送方连续接受三个重复确认时,执行乘法减小算法,把ssthresh减半,接下来不执行慢开始

       2.考虑到如果是网络拥塞就不会出现好几个重传回复,所以发送方认为没有出现拥塞,此时不执行慢开始,而是把cwnd设为ssthresh的大小,然后执行避免拥塞算法

## 常见的端口

     20     21     ftp

     22     ssh

     23     telnet

     25     smtp     简单邮件传输协议

     80     http

ssh与telnet

ssh与telnet都是基于tcp/ip的可以远程登录另一台主机的协议

不同点在于:

ssh 端口号 22 telnet是23

ssh加密传输 telnet是明文传输

ssh使用公钥对访问服务器的用户进行身份验证,telnet没有

## socket

标识网络进程    ip+协议+端口

网络中的进程通过socket来通信

socket是个接口而不是协议    它是对tcp/ip的封装

socket连接:   一对套接字 clientSocket    serverSocket

1. socket(int domain,int type,int protocol)创建一个socket  domain 协议域  type socket类型  protocol  协议
2. bind()   分配地址  通常服务器在启动时候会绑定一个地址(ip地址+端口号),用于提供服务,客户可以通过它来连接服务器  而客户端就不用指定,有系统自动分配一个端口号与自己ip地址组合. 所以服务端在listen之前会调用bind,而客户端不会,是在connect时由系统随机生成一个
3. listen()  监听   作为一个服务器,在调用socket,bind之后会调用listen来监听这个socket,如果这时客户端调用connect发出连接请求,服务端会接收
4. connect()
5. accept()   tcp服务器依次调用socket,bind,listen之后,就会监听指定的socket地址了.tcp客户端依次调用socket,connect之后就向tcp服务端发送一个连接请求. tcp服务端监听到请求后,会调用accept来接收该请求,这样连接就建立好了,可以开始网络IO操作
6. 如何检测socket是否断开  
   心跳包
7. 发送时顺序接受时顺序乱了,原因?  
   经过的路由不同
8. http协议只是tcp协议的应用层,tcp的难点在于服务端

    建立socket连接需要一对套接字,一个位于客户端ClientSocket,一个位于服务端ServerSocket,套接字之间的连接分为三个部分:服务器监听,客户端请求,连接确认

socket是长连接,http是短连接

若双方是socket连接,可由服务器向客户端发送数据

若双方是http连接,由客户端发送请求后,服务端将数据回传给客户端

socket三次握手

当客户端调用connect时，触发了连接请求，向服务器发送了SYN J包，这时connect进入阻塞状态；服务器监听到连接请求，即收到SYN J包，调用accept函数接收请求向客户端发送SYN K ，ACK J+1，这时accept进入阻塞状态；客户端收到服务器的SYN K ，ACK J+1之后，这时connect返回，并对SYN K进行确认；服务器收到ACK K+1时，accept返回，至此三次握手完毕，连接建立。

### socket流程

服务器：

1.创建一个socket

2.bind一个句柄（端口和地址）

3.listen设置监听的客户数

4.accept客户端的句柄

5.读写。

客户机：

1.创建一个socket

2.connect到服务器

3.读写操作。

## 五元组

协议     端口号     IP     端口号     IP

## tcp的状态

    Listening     侦听远方的tcp端口连接请求

    syn\_sent     客户端,syn\_sent状态

    syn\_received     服务端

    established     打开的链接

    fin-wait-1     主动关闭端调用close(),其tcp发出fin主动关闭连接,进入fin-wait-1

    fin-wait-2     主动关闭段接到ack后,进入fin-wait-2

    close-wait    被动关闭端接收到fin后,发出ack回应,进入close-wait

    last-ack     被动关闭端一段时间后,发送一个fin,等待对方ack

    time-wait     主动关闭端接到fin后,发送ack,进入time-wait

    closed     被动关闭端接受ack后

## TCP的Q

    syn队列     accept队列

    syn队列     保存处于SYN\_SENT与SYN\_RECV状态的请求

    accept队列     保存处于established状态的请求,但是应用层没有调用的请求

    syn到达->插入syn队列->返回syn+ack->由syn队列取出->插入accept队列

    第一次握手     当客户端syn到达时,若syn队列未满,插入 满了则丢弃

    第二次握手     服务器返回syn+ack 表示收到

    第三次握手     客户端发起ack,表示客户端已经准备完毕,此时服务器将连接从syn队列移至accept队列,三次握手完成

    若是服务进程处理的很慢,导致大量的请求放到accept队列,会将accept队列阻塞,当accept队列满了之后,即使client传来ack,服务器也不会响应     此时,server通过tcp\_abort\_on\_overflow来决定返回值,0表示丢弃,1表示发送RST通知给client,client显示read timeout或者connect reset by peer

# 数据库

## 范化

范化是在识别数据库中的数据元素，关系，以及定义所需的表和各表中的项目这些初始化工作之后的一个细化过程

常见的有1nf 2nf 3nf bcnf 4nf

### 1nf 第一范式

数据库表的每一列都是不可分割的，同一列不能有多个值 即 一个实体的某个属性不能有多个值或者不能有重复的属性  如果出现重复的属性，可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多

第一范式就是要求属性值不可再分成更小的部分，即属性不能是属性组合   就是无重复列

id 电话 --> id 手机 台式   1nf

### 2nf 第二范式

第二范式是在第一范式的基础上建立起来的，第二范式要求每个实例可以被唯一区分 通常是加上id

如果关系R为第一范式，并且r中的每个非主属性完全依赖于r的某个候选键，称r为第二范式

如果a是r中的一个候选键，则称a为r的主属性

注：每个非主属性都需要依赖于主属性

学号 课程号 成绩 学分    关键字为（学号，课程号） 但是非主属性学分仅依赖于课程号，对关键字是部分依赖，而不是完全依赖  因此需要拆表

### 3nf 第三范式

关系r属于第二范式，且每个非主属性都不传递依赖于r的候选键 则为第三范式

学号 姓名 课程号 成绩     姓名无重复

两个候选码 学号，课程号  姓名，课程号

学号->姓名  学号，课程号->成绩  姓名，课程号->成绩

唯一的非主属性成绩不存在部分依赖，也不存在传递依赖

### bcnf

关系模式r为第一范式，且每个属性不传递依赖于r的候选键则为bcnf

### 数据库范式

    1NF:每一条满足原子性,不可再分解

    2NF:每一条满足唯一性,记录有唯一标识

    3NF:字段不冗余

## sql语句应该考虑哪些安全性

防止sql注入,对特殊字符进行转译,过滤

使用最小权限原则,为不同的操作或用户建立账户

sql出错时,不要把数据出错的信息暴露在客户端

## 优化sql的方法

    打字段时设置为not null

    使用join代替子查询

    避免使用select \*

    用exists 代替in  用 not exists 代替not in

    使用union代替手动创建临时表

    尽量减少使用like与通配符

    使用事务与外键

    使用索引

    生成缓存

## mysql数据库引擎

    myisam  isam  innodb cvs

### myisam与innodb的区别

    基本差别:myisam不支持事务,innodb支持  myisam的性能较强

    myisam 适用于频繁的查询应用;表锁,不会出现死锁;适合小数据,小并发

    innodb  适合插入,更新较多的应用,行锁;适合大数据,大并发

    innodb与myisam最大的区别:支持事务     采用了行锁

## 锁

    数据库是一个多用户使用的共享资源,当多个用户并发的存取数据时,在数据库中会产生多个事务同时存取的情况,此时     需要加锁来对数据库进行控制

    基本锁:行级锁,表级锁

## 索引

索引是一种特殊的查询表,b+树,索引是唯一的,创建索引允许一~多个列,缺点是录入减慢,尺寸加大

    b树     b+树     哈希索引

### b树/b-树     --     二叉/多路搜索树

         从根节点开始,对节点内的关键字有序进行二分查找,如果命中则结束,否则进入查询所属的儿子节点,直到对应儿子指针为空,或者为叶子节点

### b+树     --     b-树的变体 多路搜索树

         b+树只有到达叶子节点才命中       非叶子结点仅有索引作用,不包含实际的值

         有k个叶子节点必然有k个关键字

         所有叶子节点构成一个有序链表

### b树与b+树的区别:

         B+树在内部节点上不保存数据信息,在内存页可以有更多的key.     数据存放的更加紧密,具有很好的空间局部性,因此访问叶子节点上关联得数据有更好的缓存命中率

         B+树的叶子节点都是相链的,对整个树的遍历只需要一次遍历叶子节点即可.而且数据顺序排列,且相连,便于区间查找与搜索.     而B树需要每一层递归遍历,相邻元素可能在内存中不相邻,缓存命中没有B+树好

         B树每个节点都包含k/v,经常访问的元素可能离根节点近,访问迅速

### 哈希索引

         哈希索引就是采用一定的哈希算法,把key值转换成新的哈希值,只需要一次哈希算法即可立刻定位到相应位置

索引有B+树,哈希索引,MySql里常用为B+树

哈希索引:哈希索引就是把键值根据哈希算法换成新的哈希值,只需一次计算即可定位

1. 但是对于其值是根据链表查找

2. 因为索引为计算后的哈希值,因此无法用于排序

3. 只支持等值查询,不支持范围查询

4. 存在哈希冲突的问题

B+树索引:B+树是一个平衡多叉树,根节点到叶子节点高度差值不超过1,同级间有指针相互链接

1. 在B+树中,所有记录节点按键值大小顺序排在同一层

2. B+树中,常规检索,根节点到叶子节点搜索效率相当,无大波动

3. 可以基于索引顺序扫描

NoSql索引:

NoSql的索引采用B(B-)树

原因:

B+树与B树最大的区别在于:B+只有叶子节点存放数据,其余节点存放索引

B树是每个索引点都会有个Data域,因此更适合nosql使用

### 索引优化

#### 联合索引

         最左前缀     如果使用了最左前缀,但是顺序颠倒,是可以使用索引的

#### 前缀索引

#### 优化策略

         最左前缀匹配原则

         主键外键一定要建立索引

         对于where on groupby 出现的列使用索引

         为较长的字符串使用前缀索引

         对于like查询 %放在前面就不能使用索引了     因此需要%放在后面

         尽量扩展索引,而不是新建索引

## 数据库的游标

    游标是数据库中的指针,尽管能遍历所有的行,但是一次指向一行,是一种从包含多条记录结果集中每次提取一条记录的机制

    sql的游标就是一种临时的数据库对象,可以存放数据库表中的数据行副本

    游标的常见用途是保存查询结果,结果集由SELECT生成,如果处理过程需要重复使用一个记录集,可以创建游标使用多次

## 事务

## 隔离

    未提交读     都不能避免

    已提交读     避免脏读

    可重复读     避免脏读,不可重复读

    可序列化     避免脏读,不可重复读,幻读

### 脏读

一个事务正在对一条记录做修改,在提交前,另一个事务来读取这条数据,第二个事务读取到的就时脏数据

### 不可重复读

一个事务正在读取的某些数据发生了改变,或者记录被删除     这种现象叫做不可重复读

### 幻读

  一个数据按照相同的查询条件读取以前的数据,却发现了其他事务插入的新数据叫做幻读

## 注

1. innodb的行锁是通过索引上的索引项来实现的     只有通过索引检索数据,innodb才会使用行锁,否则则会使用表锁

# Http

## http请求的过程

1. 解析url    将url解析成为对应的ip地址,无端口号则默认为80端口
2. 发起TCP3次握手建立连接   客户端向服务端发送一个请求,服务端接受同意后向客户端发送一个确认消息,客户端接受确认后向服务端发送一个确认请求,开始传输数据
3. http传输(发送http请求)

## 访问页面到加载出资源经历了什么?

开始-->DNS解析-->负载均衡-->web服务器-->浏览器渲染-->结束

1. DNS解析     输入的域名通过DNS解析成ip,供用户访问

    首先会搜索浏览器自身的DNS缓存,

    如果浏览器中没有DNS缓存或者已过期,搜索hosts文件和操作系统缓存(通过hosts,可以手动为域名添加ip解析)

    如果hosts和操作系统缓存没有找到指定的记录或者已过去,则向域名解析服务器发送解析请求

    如果域名解析服务器没有记录,则递归+迭代解析(即访问其他解析服务器)     根域服务器-->.com服务器-->google.com

2.发起http请求          建立在tcp之上的按照http协议发送的一个索要网页的请求

    负载均衡     一台服务器无法支撑大量用户访问时,将用户分摊到两个或多个服务器的方法叫负载均衡     ngnix

    ngnix根据负载均衡的分配算法建立tcp连接,转发http请求,web服务器收到请求,产生响应,将网页发送给ngnix负载均衡服务器,ngnix返回给我们的浏览器

3.浏览器渲染

    根据页面内容,渲染页面

    加载静态资源

## 计算机启动全过程

BIOS     base input output system

打开计算机到计算机准备接收指令之间的时间为引导过程(boot)

1.加电

     打开电源开关,给风扇和主板通电

2.开机自检

     系统BIOS的启动代码首先要做的就是进行POST(加电自检),POST结束后,BIOS查找显卡的BIOS并调用它的初始化代码,由显卡BIOS初始化显卡

3.加载操作系统

     在POST成功后,BIOS执行最后一项,根据启动顺序,从硬盘,软盘或光驱加载操作系统

# Jvm/操作系统

## 哈希冲突

    1. 开放定址法     线性探测再散列

    2. 链地址法

    3. 再哈希法

    4. 建立公共溢出区

## jvm内存分区

    每一个应用程序都有一个jvm,而不是共享一个

    java源程序,首先通过编译器,把java语法得代码编译成jvm语法得字节码,然后jvm通过类加载,把需要的类字节文件加载到内存中

    内存分两部分:线程共享内存    线程私有内存

         线程共享内存:     堆 方法区(运行时常量池,静态变量)

         线程非共享内存:     java虚拟机栈,本地方法栈,程序寄存器

    每个线程都有一个独有的非共享内存

    java虚拟机栈与线程一个生命周期,线程私有

    程序寄存器记录线程当前执行到哪条字节码

    本地方法栈是为了native准备的

    垃圾回收主要针对堆

    永久代就是方法区的实现,在jdk8中废弃了永久代,改为使用元空间

    方法区是线程共享的,用于存放被虚拟机加载的元数据信息,如:常量,静态变量,方法区也称之为永久代     方法区在逻辑上是属于堆的一部分,但是为了与堆区分,才叫方法区

    堆空间 = 年轻代 + 老年代

    新生代:Eden+from Survivior+To Survivior

    老年代:Survivior

    元空间:MetaSpace

    使用永久代容易发生内存不够或者内存泄露的问题,经常会爆出异常OutOfMemoryError:PermGen

    元空间时方法区在hotSpot jvm中的实现,本质与永久代类似,都是对方法区的实现,不过他与永久代的区别在于:元空间不在虚拟机中,而是在本地内存

## 堆/栈

    基本数据类型,局部变量 存放在栈中,用完消失

    new创建的实例化对象与数组,存放在堆内存,用完靠垃圾回收不定期消除

    创建数组     x[3]

          1. 在栈内存开辟空间,存放x变量

          2.在堆内存开辟空间,存放int [3],获取首地址值

          3.堆内存中的地址值给x,即x指向int[3]

     栈:

         函数中定义的基本类型变量,对象的引用变量都在栈中分配

         栈内存特点:     数据一执行完毕,变量立即释放,节约内存空间

         栈内存中的数据,没有初始值,需要手动设置

     堆:

         堆内存用来存放new创建的对象和数组

         堆内存中所有实体都有内存地址值

         堆内存中的实体都是用来封装数据的,有初始值

         堆内存中的实体不再被指向时,JVM启动垃圾回收机制,自动清除

     方法区:

         所有线程共享的区域,存放常亮,静态变量等     习惯叫永久代

     栈是一块和线程紧密相关的内存区域,每个线程都有自己的栈内存,用于存放本地变量,方法参数和栈调用,一个线程中存储的变量对于其他线程是不可见的.而堆是所有线程共享的一片公共内存空间,对象都在堆里创建,为了提升效率会在自己的栈里弄一个缓存,如果多个线程使用该变量就会发生问题,使用volatile可以使线程从主存中读取

## 垃圾回收算法

    意义:不必考虑内存管理,对象不再有作用域,只有对象的引用才有作用域     垃圾回收可以有效地防止内存泄露,有效使用空闲内存

    垃圾回收机制的算法:

         发现无用信息对象     回收被无用信息对象占用的内存空间

### 引用计数法

         每个对象实例都有一个引用计数,当一个对象被创建时,将该对象实例分配给一个变量,变量设置为1,任何其他变量赋值为这个对象的引用时,计数+1,当对象实例的某个引用超过了生命周期或者被设置为一个新值时,计数器减一     任何计数器为0的被当做垃圾回收

         缺点:     无法检测出循环引用

### 标记-清除算法

         将垃圾分成了两个阶段,标记阶段和清除阶段

         在标记阶段,通过根对象,标记所有从根节点开始的可达对象,未被标记的对象就是垃圾

         清除阶段,清除未被标记的对象

         缺点:垃圾回收后可能存在大量的内存碎片,因为对象所占的空间是固定的

### 标记压缩清除法(java中老年代使用)

         标记清除阶段如上

         压缩阶段:     将标记的对象集中放在一起,然后再去清除,这样不会产生内存碎片

         但是压缩占用了系统的消耗,如果标记的对象过多,损耗会很大,因此适合标记对象较少的情况

### 复制算法(java中新生代采用)

         核心思想是把内存空间分为两块,同一时刻只使用其中的一块,在垃圾回收时将正在使用的内存中存活的对象复制到未使用的内存中,然后清除正在使用的内存块中的所有对象

         存活对象如果较多,效率会比较差,而且这样内存折半     但是这样不会产生内存碎片

### 分代法(java堆采用)

         按照生命周期长短特点进行分块,然后每块内存区根据特点,进行不同的回收算法,提高效率

         jvm就分为新生代和老年代     新生代使用复制算法     老年代使用标记压缩清除

         所有的新生成的对象放在年轻代

         在年轻代经历了几次垃圾回收仍然存活的对象会被放入中老年代

         持久代存放静态文件,

### 分区算法

         把整个空间分为连续的不同小区间,每个小区间独立使用,独立回收,好处是一次可以控制回收多少个小区间

## linux文件读写

### 缓存

缓存是用来减少高速设备访问低速设备平均时间的组件,文件读写会涉及到内存与磁盘,内存的操作速度远远大于磁盘读写,所以如果每次调用read,write取操作磁盘,一方面速度慢,另一方面会降低磁盘的使用寿命,因此不论读写,操作系统都会把数据缓存起来

### 页缓存

页缓存是介于内存与文件之间的缓冲区,实际上是一块内存区域,所有的IO都直接与页缓存交互     操作系统通过一系列的数据结构(inode,address\_space,struct page),实现将一个文件映射到页的级别,一般来说,文件读写的优化就是对页缓存使用的优化

### 脏页

如果页缓存与对应文件区域内容不一致,该页缓存称为脏页     对页缓存进行修改,或者新建页缓存 只要磁盘没刷新,都是脏页

     查看页缓存

 free(cached的那一列)     查看/proc/meminfo(Cached  Dirty)

### 读文件

         用户发起read请求

         操作系统中查找页缓存

              若未命中,则产生缺页异常,然后创建页缓存,从磁盘读取相应页填充页缓存

              若命中,则直接从页缓存中返回要读取的内容

         read调用完成

### 写文件

         用户发起write请求

         操作系统查找页缓存

              若未命中,产生缺页异常,然后创建页缓存,将用户传入的内容写入页缓存

              若命中,则直接将用户传入的内容写入页缓存

         write调用完成

         页缓存被修改为脏页,操作系统有两种机制将脏页写回磁盘

              手动调用fsync()

              由pdflush()进程定时将页缓存写回磁盘

### 文件读写优化

         不同的优化方案适用于不同的使用场景     比如文件的大小,读写的频率

         主要考虑两点:

              最大化利用页缓存          尽量让每次IO命中页缓存,比磁盘读写快很多

              减少系统api调用次数          系统api指read与write

## 操作系统调度进程

### 先来先服务

每次调度从就绪的队列中选择一个最先进入队列的,为之分配处理机,使之投入运行

### 短作业优先

从就绪队列中选一个估计时间最短的进程

### 非抢占式优先权

把处理机分给高优先级的进程后,进程持续直到运行完毕

### 抢占式优先权

执行期间有一个更高优先级的,当前进程停止,将处理机分配给高优先级的进程

### 时间片轮转

        所有就绪的进程按照先来先服务的原则排成一个队列    每次对队首进行执行一个时间片,然后移到队尾,

## 同步异步 阻塞非阻塞

### 同步异步

        同步和异步关注的是消息通信机制

        同步,就是在发出一个调用时,没得到结果之前,该调用不会返回,但是一旦调用返回,就会得到返回值,如果有多个任务发生,这些任务必须逐个进行,一个事件或者任务的执行会导致整个流程的暂时等待,这些事件没有办法并发的执行

        异步是在调用发出之后,调用就返回了,没有返回值结果    也就是说调用者不会立即获得结果 如果有多个任务同时发生,可以并发执行,一个任务不会导致整个流程的等待

### 阻塞非阻塞

        阻塞非阻塞关注的是程序在等待调用结果时的状态

        阻塞调用是指调用结果返回之前,线程会被挂起,该线程只有在得到结果之后才会返回 当一个任务在执行过程中,他发出一个请求,但是由于该请求需要的条件不满足,就会一直等待,知道条件满足

        非阻塞调用指在不能立刻得到结果之前,该调用不会阻塞当前线程 当一个任务在执行过程中,发出请求,如果请求需要的条件不满足,会返回一个信息告知,而不会一直等待

         比如读文件时,阻塞会一直等待,非阻塞会返回一个信息

### 阻塞IO与非阻塞IO

         当用户线程发起一个IO请求的时候,内核会查看请求的数据是否就绪,对于阻塞IO,如果数据没有就绪,则会一直等待,直到数据就绪,     而对于非阻塞IO,如果数据没有就绪,则会返回一个标识信息告知用户线程.     当数据就绪之后,会把数据拷贝到用户线程,完成一个IO请求     阻塞和非阻塞IO的区别在于开始判别数据是否就绪时一个会等待,一个会返回信息

         java中传统IO都是阻塞IO

### 同步IO与异步IO

          对于同步IO:当用户发出IO请求之后,如果数据没有就绪,需要用户线程或者内核不断轮询数据是否就绪,当数据就绪时,再将数据拷贝至用户线程

         对于异步IO:只有IO请求操作是由用户线程发出的,其他的操作交由内核,然后告知用户线程IO操作完成

         同步IO与异步IO的区别在于数据拷贝阶段是由内核还是用户线程完成,异步IO必须有操作系统底层支持

### 五种IO模型:

              阻塞IO     非阻塞IO     多路复用IO(NIO)     信号驱动IO     异步IO

### IO与NIO

    io: 其实意味着 不断地把数据搬入搬出缓冲区,     比如,用户发起了读操作,导致"syscall read",就会把数据搬入到一个buffer中,发起写操作也是如此

    IO:以流为基础进行输入输出,数据串行化写入写出     IO操作包括:对硬盘的读写,对socket的读写,对外设的读写

    NIO:non-blocking IO     为所有的原始类型提供buffer缓存支持     非阻塞,多路复用

#### IO与NIO的区别:

         IO面向流,     NIO面向块缓冲区

              IO面向流的操作是一次一个字节的处理数据,一个输入流产生一个字节的数据,一个输出流消费一个字节,会导致读写效率不佳

              NIO面向块的操作在一步中产生或者消费一个数据块,按块处理比按流处理快很多,同时数据会读取到稍后处理的一个缓冲区,采取了预读的方式,当读取某一部分数据时,会猜测下一步要读取的先缓冲下来

         IO是阻塞的     NIO非阻塞

              对于传统IO,线程在调用read()和write()时,线程被阻塞,直到一些数据被读取,或者完全写入.该线程在期间不会做任何事      一旦readLine()返回,则说明整行已经被读取

              对于NIO,使用一个线程发送读取数据请求,在没有得到响应之前,线程是空闲的,此时可以去处理其他任务,不用像IO中等待

#### NIO与IO的适用场景

         如需管理同时打开的成千上万条连接,连接每次发送少量数据,使用NIO

         如只有少量的连接,每次大量数据,使用IO

### NIO中的概念

#### 通道:Channel

              类似于流,但是流只能同步读写,会阻塞, 通道符可以异步读写,不会阻塞

              通道是双向的,流是单向的     (InputStream只能读,channel可读可写)

              FileChannel(文件读写)     SocketChannel(tcp连接)     ServerSocketChannel(socket监听)     DatagramChannel(udp协议)

#### 缓冲区:buffer

              通道的数据总是先读到一个buffer或者从一个buffer写入

              buffer实际上就是一个容器,是一个连续数组

              Channel提供从文件,网络中读取数据的渠道,但是读取或者写入的数据都要经过Buffer

              客户端发送给服务端:

                   客户端写入buffer,buffer写入通道 通道->通道 通道写入buffer buffer写入服务端

               ByteBuffer(网络中最常用)     Int/Char/Long/Double/Float/Short+Buffer

#### 选择器:Selectors

              selectors就是用来轮询每一个注册的Channel,一旦发现注册的Channel有事情发生,便获取事件进行处理

              selectors允许一条线程去监控多个channels的输入,可以向一个selector上注册多个channels,然后调用select()方法判断是否有新的连接进来

## jstack

    jstack是java虚拟机自带的一种堆栈跟踪工具,用于生成java虚拟机当前时刻的线程快照     jstack [-l]  pid

    可以定位到线程堆栈,根据堆栈信息可以定位到具体代码,进行JVM调优

    找某个java进程中最耗费cpu的线程并定位堆栈信息

          1. 找pid     ps -aux|grep xxx     得到pid  -->21711

          2.找进程内最耗cpu的线程     top -Hp pid(21711)       -->21714

          3.取得该线程pid的十六进制数字     print "%x/n" pid(21714)     -->54ee

          4.使用jstack输出线程的堆栈信息  jstack pid(21711)|grep 54ee     -->

         "PollIntervalRetrySchedulerThread" prio=10 tid=0x00007f950043e000 nid=0x54ee in Object.wait()

       代表CPU消耗在这个类的Object.wait()

## jvm参数

|  |  |
| --- | --- |
| -Xms20M | 堆容量最小值为20,必须M为单位 |
| -Xmx20M | 堆容量最大值为20,-Xms与-Xmx设置一样可以避免堆自动扩展 |
| -Xss128k | 虚拟机栈大小 |
| -Xoss128k | 本地方法栈大小 |
| -verbose:gc | 输出虚拟机GC的详细信息 |
| -XX:PermSize=10M  -XX:MaxPermSize = 64M | 永久代容量 |

|  |  |
| --- | --- |
| -Xnoclassgc | 关闭对类的垃圾回收 |
| -XX:+TraceClassLoading  -XX:+TraceClassUnLoading | 查看类的加载/卸载信息 |
| -XX:NewRatio=4 | 表示设置年轻代:老年代(不含永久代)为1:4,即年轻代占堆的1/5  新生代 eden |
| -XX:SurvivorRatio=8 | 表示两个永久代和新生代的比值 |
| -Xmn20M | 年轻代大小为20M |
| -XX:PrintGC  -XX:PrintGCDetails | 打印gc的信息 |

### 注

         同步IO异步IO与阻塞IO非阻塞IO是两个不同的概念

              阻塞IO与非阻塞IO反映在当用户请求操作IO时,数据没有就绪,是用户线程会一直等待,还是会返回标记这一点上

              同步IO与异步IO反应在拷贝阶段由用户线程还是内核完成这一点

# 多线程

## 并行与并发

    所有的并发操作都会有排队等待,唤醒,执行的操作   因此并发只是宏观上的概念 在微观上他们是被序列处理的  只不过资源会通过 **时间片轮转** 使得程序不会在一个上被阻塞,所以从宏观上来说,是同时到达的请求同时在被处理

    并发:     并发的实质是一个/多个cpu在若干个程序之间被复用, 并发是对有效物理资源强制行使多用户共享来提高效率     (UG的反复横跳)

    并行:     并行的实质是两个或两个以上的事件或活动同时发生, 多个程序同一时间在不同cpu上同时发生                         (米波的五狗齐飞)

    并发是在同一个cpu通过时间片轮转,近似认为同时进行

    并行是在不同cpu同时进行

    举例:

         并发     一个人(cpu)同时喂A,B(线程)吃饭

         并行     两个人同时,一个人喂A吃饭,一个人喂B吃饭

    并行需要两个或两个以上的线程跑在不同处理器上     并发可以跑在一个处理器上通过时间片进行切换

### 为什么用并行,并发?

         很多任务是需要花费时间的,如文件读写,网络下载     如果顺序执行会使得CPU处于闲置状态,效率低下     并发并行会极大的缩短不必要的耗时

## 线程与进程

    线程状态:     新建-->就绪-->运行-->阻塞(-->就绪)     (运行-->)死亡

    线程与进程都是由操作系统所体现的基本单元,系统利用基本单元实现并发性

    一个应用程序可以有一个到多个进程,而一个进程可以有一个到多个线程     线程是轻量级进程

    进程     是一个实体,每一个进程都有自己独立的地址空间,包括文本区域,数据区域以及堆栈,文本区域存储处理器执行的代码,数据区域存储变量和进程执行期间使用的动态分布的内存,堆栈区域存储着活动过程调用的指令和本地变量     就绪,运行,阻塞     就绪:获取到了除CPU外的所有资源     运行:获取到了CPU分配的资源,开始执行     阻塞:如等待I/O

    线程     一个进程中可以包含多个线程,且至少有一个线程, 线程可以利用进程所拥有的资源, 通常把进程作为分配资源的基本单位,把线程作为独立运行和独立调度的基本单位     因为线程比进程小,基本上不拥有系统资源,调度开销会小的多,因此可以高效的提高系统多程序间的并发

    程序     指令和数据的有序集合,本身没有含义,是个静态的概念

### 区别

         线程是CPU独立分配和独立调度的基本单位     进程是资源分配的基本单位

         进程在运行中有多个独立的内存单元     一个进程中的多个线程共享堆内存

         资源分配给进程     同一进程的所有线程共享代码段,数据段,堆内存,  但是每个线程有自己的栈,用来存放局部变量和临时变量

         真正在处理机上运行的是线程

    比较:(为什么多线程而不是多进程)

         系统开销:     因为在创建,撤销进程时,系统要为之分配或回收资源,如内存空间,IO设备  因此远大于在创建,撤销线程时的开销   类似的在进行进程切换时,线程切换仅需要保存和设置少量寄存器的内容

         同一进程中的多个线程具有相同的地址空间,同步与通信十分容易

### 进程间通信

         进程间的通信使用管道,消息队列,套接字,信号量,Stream    socket和Stream支持两个不同主机间的通信

         每个进程都有自己的内存空间,任何一个进程的全局变量在另外一个进程中是看不到的,因此进程之间的交换数据必须通过内核,在内核中开辟一个缓冲区,进程A把数据从用户空间拷贝到内核缓冲区,进程B再从缓冲区把数据取走,内核提供的这种机制叫做进程间的通信

         不同进程间的通信本质:进程之间可以看到一份公共资源,提供这份资源的形式或者提供者不同,造成了通信方式的不同

#### 管道:pipe

               是半双工的,也就是数据只能单向流通 具有固定的读端和写端

               只能在父子进程或者兄弟进程之间通信

               可以看做是一个特殊的文件,对于他的读写使用read,write,但是不属于任何文件系统,并且只存与内存中

#### 命名管道:FIFO

                是一种文件类型

                FIFO可以在无关的进程之间交换数据

                以一种特殊设备文件形式存在于文件系统中

#### 消息队列

                 消息的链接表,存放在内核中,一个消息队列有一个标识符(队列ID)标识

                 消息队列面向记录,其中的消息具有特定的格式和优先级

                 独立于发送和接受进程,进程即使终止,消息队列的消息也不会删除

                 可以实现消息的随机查询

#### Socket

## 闭锁,栅栏,信号量

     闭锁CountDownLatch     栅栏CyclicBarrier     信号量Semaphore

### 闭锁

用于一组线程等待一个外部事件的发生,这个事件发生之前这些线程被阻塞,等待控制程序打开闭锁,然后这些程序同时执行

闭锁     CountDownLatch cdl = new CountDownLatch(3)     cdl.await()     cdl.countDown()     每次调用countDown()时,计数器减一,当计数器为0时,打开锁

### 栅栏

是一组程序相互等待,直到所有线程达到某一点打开栅栏

         栅栏可以在达到指定状态时调用runnable实现方法,(新的线程,且只调用一次)

  栅栏     CycliBarrier cb = new CycliBarrier(3,new Runnable())     cb.awit()     Runnable只执行一次     每次调用await()时,计数器加一,计数器为参数时,打开栅栏

     使用闭锁和栅栏可以使多个线程在同一时间争夺资源

闭锁与栅栏的区别在于 闭锁只能执行一次 栅栏可以多次使用     可以把闭锁看做一次性的栅栏

### 信号量

信号量和synchronized差不多,不过synchronized只可以一个线程拿到锁,信号量支持多个线程同时取锁

    信号量     Semaphore sema = new Semaphore(3)     sema.acquire()     sema.release()     锁的内容在acquire与release之间     允许最多三个线程同时访问锁的内容

## 线程的实现/线程池

    继承Thread     实现Runnable     实现Callable

    实现Callable可以有返回值    <--重写call()     使用Future来获取     future = pool.submit(new Callable(){})

### 线程池

#### corePoolSize

核心线程数 代表可以并发执行的线程总数,当当前线程数小于核心线程数时,无论是否超过空闲时间,线程不会退出,直到线程数 大于核心线程数

#### queueCapacity

阻塞/任务队列 当核心线程数达到最大时,新任务会放在队列中排队等待

1. ArrayBlockingQueue 基于数组的队列,需要指定大小 因此会存在阻塞队列满的情况

2. LinkedBlockingQueue 基于链表的队列,默认为Integer.MAX\_VALUE 基本不会存在阻塞队列满,即不会达到最大线程数

3. synchronousQueue 不保存新提交的任务,而是创建一个新的线程执行

#### maxPoolSize

最大线程数 当前线程>核心线程数时,且阻塞队列满,会创建新的线程执行任务

当前线程>最大线程数时,且阻塞队列满,异常

在此处需要注意的是 阻塞队列为Link

#### keepAliveTime

空闲时间 当线程空闲时间达到keepAliveTime时,线程会退出,直到corePoolSize

### 线程池创建过程

对于线程池,我们在服务端进行一次初始化即可

1.每次来新的任务,先查看当前核心线程是否空闲,空闲则使用该线程,

2.再比较当前线程数与核心线程数,若当前线程数小于核心线程数,则创建线程

3.若当前线程数大于核心线程数,则将任务放入任务队列中

若任务队列未满,继续将任务存入队列

若任务队列满,则创建新的线程 此时若线程达到最大线程数时,仍然传来任务,则会采用任务拒绝策略

期间若是核心线程空闲,则从任务队列中取出任务,并执行

### 线程池数量的设置

    工作性质 IO密集型,CPU密集型

    如果是CPU密集型,需要设置较少的线程数,比如CPU数目+1     如果是IO密集型,应该设置可能多的线程数,由于IO不占CPU,所以不能让CPU闲下来,当然,如果线程数目过多,线程切换所带来的开销又会对系统的响应时间有影响

### 线程状态

    java中线程有六种状态:

 new      runnable     blocked     waiting     time\_waiting     terminated

         new 指的是新建了线程 但是线程还未启动     没调用start()

         runnable     可运行的,运行中

         blocked     阻塞状态

         waiting     等待状态     wait()     join()

         timed\_waiting    有限时间的等待  sleep()     wait(time)     join(time)

         treminated     终止/死亡     也就是线程的run()执行完毕

## Monitor(不太清楚)

线程之间的互斥与通信,主要靠的是monitor

进入区:     线程通过synchronized要求获取对象的锁,如果对象未被锁住,则冲入拥有者,否则在进入去区等待,一旦对象的锁被释放,立即争抢

拥有者:     某一线程成功竞争到对象

等待区:     线程通过对象的wait,释放对象的锁,进入等待区

monitor就是拥有者     一个monitor在个时刻只能被一个线程拥有

## ThreadLocal

    线程本地变量/线程本地存储

    ThreadLocal为多线程并发问题提供了新的思路,用于解决多线程访问资源时的共享问题

    为变量在每个线程中都创建了一个副本,每个线程可以访问自己内部的副本变量,也就是把共享的资源生成副本分别放到每个线程中,这样就使得各个线程不共享资源,解决了同步问题

    ThreadLocal变量通常会用private static修饰,在一个线程结束后,使用的所有ThreadLocal副本会被回收

    ThreadLocal适用于每个线程需要自己独立的实例且该实例需要在多个方法中被使用,也就是变量在线程间隔离但是在方法中共享的场景

    例:

         数据库连接

         如果是静态的,在多线程资源时会导致不安全

         如果每次都生成一个连接,则会由于不断创建,销毁导致服务器压力大

         使用ThreadLocal可以在每个线程中创建一个副本,线程之间互不影响,会导致资源消耗较大,但是没有过多的创建连接

## 单例模式

### 饿汉模式

         立即加载,使用类的时候对象已经创建完毕

         private static C c = new C()     return c

### 懒汉模式

         延迟加载,调用get()时才创建

         private static C c

         get()

             if(c==null)     c=new C()     //多线程中是不安全的

         改良:

#### 同步方法

                   private static C c

                   synchronized get()

                       if(c==null)     c = new C()

#### 同步代码块

                   synchronized(C.class){

                       if(c==null)     c = new C()     }

#### 双重检查机制

                   private volatile static C c;

                   synchronized(C.class) {   if(c==null) c=new C()     }

## join

让主线程等待子线程执行完毕后才能继续执行

class Father{

    public void run(){

        Son s = new Son()

        s.start()

        s.join()               F等待S执行完后再执行

}}

join的底层是wait()     会释放锁

## COW(CopyOnWriteArrayList)

     CopyOnWrite容器: 写时复制的容器,往一个容器添加元素的时候,不直接往当前容器添加,而是现将当前容器进行Copy,给新容器里添加元素,添加完之后,把原容器的引用指向新容器.     好处是可以对CopyOnWrite容器进行并发读,而不需要加锁,是一种读写分离的思想,读与写使用不同的容器

#### CopyOnWriteArrayList的实现原理

         向ArrayList里添加元素时,在添加时是需要加锁的,否则会复制多个副本

         在读取元素时,不需要加锁,如果此时有多个线程向ArrayList添加元素,读还是会读取到旧数据,因为写的时候不会锁住旧的ArrayList

#### CopyOnWrite使用场景

         适用于读多写少的并发场景,比如白名单,黑名单,商品显示

#### COW的缺点

         内存占用:     因为写时复制,所以会有两份内存占用,可能引起频繁的垃圾回收,数据量较多,用ConcurrentHashMap

         数据一致性问题:     只能保证数据的最终一致性,不能保证实时一致性,如果希望存取的立马获取,不适用COW

## ConturrentHashMap

🡪[Click](#_HashMap__)🡨>

## Interrupted()与IsInterrupted()

    Interrupted 与 IsInterrupted 的区别在于 前者可以消除中断状态而后者不会

    java多线程的中断机制是用内部标识来实现的,调用Thread.interrupt()来中断一个线程就会设置中断标识为true,在中断线程中调用Thread.Interrupted()检查中断标识时,会把中断标识清零     而isInterrupted()来查询线程是否中断时不会清零

holdsLock()     可以检测是否拥有锁

## execute()     submit()

    execute()返回类型是void     submit()可以返回Furure对象

## 忙循环

    就是用循环让一个线程等待,不会放弃CPU控制,目的是保留CPU缓存,在多核系统中,一个等待线程醒来时可能会在另一个内核中运行,这样会重建缓存,为了避免重建缓存和减少重建的时间就可以使用忙循环

## 多线程实用方法:

    1.给线程起有意义的名字,方便问题追踪

    2.缩小同步的范围     最低界限使用同步和锁,缩小临界区     (同步方法欧式抛出异常,则会释放锁)

    3.多用同步类,少用wait与notify      同步类(CountDownLatch,Semaphore,CyclicBarrier)

    4.多用并发集合,少用同步集合

              同步集合     HashTable                      Vector

              并发集合     ConcurrentHashMap     CopyOnWriteArrayList

## 注

1. 无法强制启动一个线程,正如无法强制垃圾回收 System.gc()并不能保证成功
2. i++与++i

    i++与++i都是非线程安全的,需要加上synchronized

1. 一些函数

isAlive()     判断线程是否处于活动状态

getId()

interrupt()       仅仅是打了一个停止的标记,并没有真的停止线程

thread.interrupted()     测试当前线程是否已经中断,会清除中断状态     也就是如果第一次中断,调用interrupted(),之后没有中断,则第二次return false

isInterrupted()              测试是否已经中断,不会清除中断状态

yield()      线程让步   放弃当前的CPU资源,把它让给其他任务,但是也有可能刚放弃就获取

setPriority()     设置线程优先级     1~10     不在范围内 IllegalArgumentException     优先级还具有随机性,总是优先获取CPU资源,但是不能保证一定会先执行完

1. 守护线程:     当进程中不存在非守护线程时,守护线程自动销毁     典型的守护线程:垃圾回收线程
2. 非线程安全问题仅存在于实例变量,(即共享资源)中     对于方法内部的变量,不会存在线程安全的问题
3. 为什么 wait() notify() notifyAll() 是在object中而不是Thread中

     一个很明显的原因是java中的锁是对象级而不是线程级,wait在使用的时候会释放锁,每个对象都有锁,通过线程获得.

1. 为什么wait和notify要在同步块中?

      1.java API强制要求          2.避免wait与notify产生竞争条件

1. FutureTask FutureTask是一个可以取消的异步运算,只有运算完成的时候才会获得结果,如果运算未完成,调用get()将会被阻塞

# 数据结构

## HashMap     ConcurrentHashMap     HashTable

通过将key,value保存到entry中,通过拉链法插入哈希桶     如果数量达到临界值(默认为诶哈希桶长度的0.75,则扩容)

    但是hashmap是线程不安全的,因此有了concurrent包的concurrentHashMap

### HashMap原理

         获取key的hash值

         返回在hash桶里的位置

         遍历当前位置后面的元素

              如果有相同的key,则替换value

              否则放入entry(key+value)          放入entry时会检测是否达到临界值,判断是否扩容

### ConcurrentHashMap原理

         获取key的hash值

         返回hash的位置

         concurrentHashMap将hash桶进行了分段,每个片段上都有锁,然后再片段上进行插入,获取锁

         遍历并插入          是否扩容

### hashmap与concurrenthashmap的区别

         concurrentHashMap对桶进行了分段,由一个个Segment组成,concurrentHashMap就是一个Segment数组,Segment继承ReentranLock来进行加锁,每次锁住一个Segment,保证了Segment的线程安全,默认concurrentHashMap有16个Segment,不扩容    而hashmap没有加锁

### concurrentHashMap与HashTable的区别

         都可以用于多线程,

         hashTable锁定整个map,当HashTable增加到一定的时候,性能会急剧下降,因为迭代时被锁定很长时间

         而ConcurrentHashMap引入了分割,不论多大,仅需要锁定map的某个部分,其他线程不需要等待迭代完成就可以访问map

### 其他

    在jdk7中 hashmap采用的是 哈希桶+链表     jdk8中采用 哈希桶+链表/红黑树     冲突节点>=8时用红黑树

## TreeMap

     TreeMap底层是红黑树,所以ContainsKey(),get(),set()都是log(n)

     TreeMap是非同步的

### TreeMap的构造函数

          1.TreeMap()     默认比较器

          2.TreeMap(Comparator comparator)     使用比较器

          3.TreeMap(Map map)     会调用putAll()把map中所有元素添加到TreeMap

    TreeMap的函数:

         firstEntry lastEntry lowerEntry higherEntry

         firstEntry()     getFirstEntry()

              firstEntry对外接口getKey(),getValue     getFirstEntry对内,还有setValue()

         firstKey lastKey lowerKey higherKey floorKey ceilingKey

         values()     返回value集合

         遍历:   int key:treemap.desendingKeySet     或者 treemap.desendingMap()

    使用o1.compareTo(o2)     进行比较器排序

## LinkedList     ArrayList 数组

    Iterator  = list.iterator()

### 数组

         数组在内存中使连续存储的,索引很快,而且赋值与修改也简单

         缺点:

              插入麻烦

              声明是需要指定长度,过长会造成内存浪费,过短会造成数据溢出

### ArrayList

         按照存储数据,动态扩容收缩     因此不需要初始化大小

         如果数组容量不足,进行扩容     Arrays.copyof

         arrayList与vector等基于数组的List可以初始化大小,减少耗时     如100W元素     arrayList()<<arrayList(100W)

         优点:     读取时,O(1)根据索引

         缺点:     添加时很慢,需要移动后面的数,且可能会扩容

### LinkedList

         优点:     添加值快,只需要更改指针

         remove的时间复杂度时O(n),因为需要比较查找

### ConcurrentModifictionException

发生在对一个Collection边遍历边修改size的操作中

    解决方案:     1.加锁          2.使用并发容器CopyOnWriteArrayList代替ArrayList与Vector

## 图

图是由顶点集合以及顶点之间关系集合组成的一个数据结构

分为有向图,无向图

图的存储结构是一个邻接矩阵/邻接表/十字链表

对于图的遍历:

深度优先遍历 -- DFS

以一个顶点为基础,总是以一个方向作为子树的根节点,其余作为叶子节点

广度优先遍历 -- BFS

以一个顶点为基础,对其进行类似于树的层序遍历(队列),直到没有未使用连接

## 红黑树

    一种特殊的二叉查找树,每个节点都有存储位表示存储的颜色,可以是红或黑

### 特性

         1.每个节点是红色或黑色

         2.根节点是黑色

         3.每个为空的叶子节点是黑色(指的是为空的叶子节点)

         4.如果一个节点是红色,子节点一定是黑色

         5.对于每个节点,从该节点到树尾的任何路径,都包含相同数量的黑色节点

    用于存储有序的数据,O(lgn)     TreeMap,TreeSet,linux虚拟内存管理都是通过红黑树

    在树的结构发生改变(删除,插入)时,会破坏4,5,需要调整使得查找树满足红黑树的条件

                    x

             a             b

         c       d

### 右旋

         右旋x就是把x变为右节点     x的左孩子->d     a的右孩子->x

                     a

             c             x

                        d     b

### 左旋

         左旋a就是把a变左右节点     a的右孩子->d     x的左孩子->a

### **插入**

        红黑树的插入会改变结构,因此需要进行调整,以满足红黑树的性质,在插入时默认为红色,因为如果是黑色的话,该条路径的黑色节点就比其他的多,不符合特性5,调整较麻烦     通过变色+旋转就可以调整红黑树

#### 五种情况

* 1. 情况1:

                   插入的是根节点     ->把根节点变为黑色

* 1. 情况2:

                   父节点是黑色节点  ->对红黑树性质无影响     不需要调整

* 1. 情况3:

                   父节点是红色节点,叔叔节点也是红色:

                        由于插入的节点与父节点都是红色,需要调整,     先将父节点与叔叔节点变为黑色,把祖先节点变为红色,再对上递归

* 1. 情况4:

                   父节点是红色,叔叔节点是黑色,且插入的是左节点

                        对祖先节点进行右旋,将父节点改为黑色,原祖先节点改为红色

* 1. 情况5:

                   父节点是红色,叔叔节点时黑色,且插入的是右节点

                        先对父节点进行左旋,变成情况4     然后对祖先节点进行右旋,插入的节点改为黑色,原祖先节点改为红色

### 删除

          对于删除操作,需要先判断删除节点有几个孩子,如果有两个,不能直接删除,而是要找到该节点的前驱(左子树的最大节点)或者后继(右子树的最小节点),将前驱或者后继的值复制到要删除的节点中,然后删除前驱或后继.     由于前驱或者后继至多有一个孩子节点,就可以吧原来的两个节点问题简化为一个孩子节点的问题

## 集合类

### Map

         键值对映射

### Iterable

         迭代器接口,是Collection的父接口,实现了Iterator接口的对象允许使用foreach遍历,所有的Collection对象都具有可遍历性     调用iterator()返回迭代器

### Collection

         List:     保持特定的顺序

         Set:     不包含相同的值

         Queue:保持队列的先进先出

             最基本的集合接口,一个Collection代表一组Object集合,他是一个接口

#### Set

    不能包含重复元素     使用equals来判别 而不是==

##### HashSet

    HashSet是Set的实现,用hash算法来存储元素,具有良好的存取和查找能力  对于存储的元素调用hashCode()取得哈希值,然后决定在hashSet中的存放位置     对于hashset来说,判别equals与hashCode

##### LinkedHashSet

      也是根据hashcode来决定存储位置,但是同时使用链表维护元素的次序     遍历LinkedHashSet时,会以添加的顺序来访问

##### SortedSet

        是一个用于实现排序的接口

##### TreeSet

         基于红黑树的排序set

#### List

##### ArrayList

##### Vector

##### Stack

Stack是vector的子类

##### LinkedList

### Queue

#### PriorityQueue

不是一个比较标准的队列,按照元素大小重新排列

#### Deque

双端队列

      LinkedList

### Map

 保存映射关系

#### HashMap

#### TreeMap

# 锁

## synchronized     与     Lock

### 锁的释放

#### synchronized

       线程执行完代码,释放锁

       线程执行发生异常,jvm让线程释放锁

#### lock

        在finally中必须释放锁,不然很容易会造成线程死锁

### 锁的获取

#### synchronized

        A获得锁,其他等待     若A阻塞,则其他一直等待

#### lock

     尝试获得锁,不必一直等待

### Synchronized的特点

    synchronized可以保证方法或者代码块在运行时,同一时刻只有一个线程运行

        普通同步方法,锁是当前实例对象

        静态同步方法,琐是当前类的class对象

        同步方法块,锁是括号里的对象

        同步代码块使用monitorenter和monitorexit实现

### synchronized的缺点

      不能响应中断

       不管读写只能有一个线程对共享资源操作,其他线程只能等待

       锁的释放由虚拟机完成,不用人工干预     好处是不会死锁 缺点是线程阻塞后其他线程会等待,影响效率

### ReentranLock与ReentranReadWriteLock

都为常用的锁,都是可重入锁

    区别:

          1. synchronized在成功完成功能或者抛出异常时,虚拟机会自动释放线程占有的锁     而Lock对象在发生异常时,如果没有主动调用unlock()释放锁,锁对象会一直持有,因此Lock需要在finally中释放锁     即synchronized自动释放锁     Lock必须手动释放锁

          2.lock接口可以通过多种方法来尝试获取锁     避免阻塞之后一直等待

           3.在读多,写少的高并发下  使用ReentranReadWriteLock分别获取读锁与写锁来提高效率,因为读锁是共享锁,写锁时排它锁     Lock可以提高多个线程读操作的效率

          4.Lock是一个接口     而synchronized是关键字

          5.Lock基于jdk     synchronized基于jvm

### Lock

         ReentranLock实现了Lock接口,内部有三个内部类     Sync     NonfairSync(非公平)     FairSync

         lock()     unlock()     trylock()

## volatile

### volatile的原理

  在java中,为了保证程序的高效,把经常会访问/使用的变量放到寄存器中,而不是内存中,使用volatile之后会使得每次都从内存中读取

### volatile与atomic

        volatile可以确保先行关系,写操作会发生在后续的读操作之前,但是不能保证原子性,如i++

        AtomicInteger类的atomic()方法可以让这种操作具有原子性,比如getAndIncrement()方法会原子性的加一

## 乐观锁与悲观锁

乐观锁与悲观锁只是一种思路,而不是具体实现

### 乐观锁

总认为不会产生并发问题,每次取数据时总认为不会有其他线程对数据进行了更改,因此不会上锁     但是在更新时会判断其他线程在这之前有没有对数据进行修改  一般使用版本号机制,CAS实现

#### CAS     compare and set

        多个线程同时更新一个变量时,只有其中一个线程能更改变量的值,其他线程都失败,失败的线程不会挂起,而是被告知失败,可以再次尝试

         CAS包括三个操作数(需要读取的内存位置V,进行比较的预期原值A,拟写入的新值B)如果内存位置V的值与预期的A相匹配,那么处理器会把该位置的值变为B,否则不做任何操作,只返回V处的值     "认为V处应该是A,如果是A,变为B,     否则不改变,只需要告诉我这个位置的值"

         每次不加锁而是假设没有冲突取完成某项操作,如果冲突失败就重试,直到成功     AtomicInteger     AtomicReference

         存在ABA问题     原值为A,线程1把A改为B之后又改为A,线程2发现还是A,但其实以及不是原来的A了,去仍然会把值改为C     使用带版本号的CAS               AtomicStampedReference

         除了依赖CPU实现带版本号的CAS,还可以借助数据库

#### 乐观锁的特点

        吞吐量高,适用于数据冲突较少的场景

        受限于外部系统,可能会有脏读     数据冲突很大的情况下,效率很低

       乐观锁主要就是解决     冲突检测 和 数据更新

### 悲观锁

 总是假设最坏的情况,每次取数据都认为其他线程会修改,所以都会加锁   synchronized,RetreenLock就是悲观锁     传统的关系型数据库就使用了悲观锁机制,(行锁,表锁,读锁,写锁 都是在操作之前加锁)

#### 悲观锁的特点

       严格避免数据混乱的问题     但是性能消耗大

## 锁的类型

### 可重入锁

synchronized,ReentranLock在执行对象中所有同步方法不用再次获得锁,即进行一次取锁后,在不释放锁的前提下,可以再次获取当前拿到的锁(防止死锁)     递归无阻塞的同步机制

#### 原理:

lockBy保存已获得锁实例的线程,在lock判断lock线程是否已获得当前锁实例,如果获得,则直接跳过while,无需等待

      lockCount记录同一个线程重复对一个锁对象加锁的次数

### 读写锁

分为读锁与写锁

读锁可以共享,多个线程可以同时拥有读锁

但是写锁却只有一个线程可以拥有,而且获取写锁的时候其他线程都释放了读锁,并且该线程获取写锁后,其他线程不可获取读锁

写锁是排它锁,读锁时共享锁

### 公平锁,非公平锁

公平锁指多个线程按照申请锁的顺序来获取锁

非公平锁不按照申请锁的顺序

非公平锁的吞吐量比公平锁大     Synchronized是非公平锁

### 自旋锁

让线程等待一段时间,不会被立即挂起,看持有锁的线程是否会很快释放锁(执行一段无意义的循环,自旋)        也称作不可重入锁

## 死锁

    死锁是指两个或多个**进程**在争夺资源的过程中的一种互相等待的现象,如果没有外力干预,都无法推进下去.

### 四个条件

         互斥资源:     一个资源一次只能被一个进程调用

         请求与保持条件:     一个进程因请求资源被阻塞时,不会释放已获得资源

         不剥夺条件:          进程已获得的资源,在未执行完时,不会被强行剥夺

         循环等待:               若干进程之间形成一种头尾相接的循环等待资源关系

### 避免死锁的方法

 组织循环等待,将系统所有的资源设置标志位,排序,规定所有进程申请资源必须以一定的顺序来操作避免死锁

# 分布式

## 大数据

### 概念

         不能使用一台机器进行处理的数据

         核心是样本=总体

### 特性

         大量性volume     单个文件级别很大

         快速性velocity    反映在数据的快速产生和数据变更上

         多样性variety      数据类型和其来源的多样化 进一步可以把数据归纳为结构化,半结构化,非结构化

         易变性

          准确性

         复杂性

### 技术

#### 数据分布在多台机器

              可靠性:     每个数据块都复制到多个节点

              性能:        多个节点同时处理数据

#### 计算随着数据走

              网络IO速度<<磁盘IO速度 大数据系统会尽量把任务分配到离数据最近的机器运行

## 分布式系统

    一群独立的计算机共同对外提供服务,但是对于系统用户来说,像是一台计算机在提供服务一样

### 几个特性

#### 分布性

              多台计算机没有主从之分

#### 透明性

              资源共享,每台计算机不仅可以使用本身的资源,还可以使用其他系统的资源

#### 同一性

              一个程序可以分布在几台计算机上并行的运行

#### 通信性

              任意两台计算机都可以通过通信交换信息

    分布式计算的核心就是利用分布式算法,把运行在单台机器上的程序扩展到多台机器并行运行,使得数据处理能力成倍增加

## hadoop

    hadoop是一个分布式系统架构     核心设计是:HDFS与MapReduce

    HDFS     分布式文件存储

    YARN     分布式资源管理

    MapReduce     分布式计算

    Hadoop要做的事情:把1PB的数据导入HDFS,然后编程人员定义好map和reduce函数,也就是把文件的行定为key,每行内容为value,然后正则匹配,进行计算,最后把结果通过reduce聚合起来返回    hadoop就是把这个程序分部到N个结点去并行的操作

### HDFS

为海量数据提供了存储     MapReduce为海量数据提供了计算

    HDFS     分布式文件系统

         block 数据块

              是基本存储单位,一般为64M     配置大的block是因为: 1.减少寻址时间     2.减少管理块的数据开销,每个块都需要在NameNode上有相应的记录     3.对块进行读写,减少建立网络的连接成本

**磁盘寻址速度远远小于数据传输速度**

              一个文件会被拆成一个个的块,然后存储在不同的机器上,若文件少于block大小,实际上占的空间为文件大小

              每个块都会被复制到多个机器,默认三份

## 负载均衡

    当一台服务器的性能达到极限时,可以使用服务器集群来提高整体性能     在服务器集群中,需要有一台服务器充当调度者,所有的请求都会首先由他接收,调度者再根据每台的负载情况将请求分配给某一台后端服务器

### http重定向

#### 过程

        用户向服务器发起请求时,请求首先被调度者截获,调度者根据资源分配策略,将选中的服务器的IP地址封装在http响应头部的location字段中,并返回302重定向     浏览器接受到重定向消息后,解析Location,并访问url

        使用http重定向时 需要一台服务器作为请求调度者,用户的一项操作需要发起两次http请求,一次向调度服务器发起,获取后端服务器的IP,第二次向后端服务器发起,获取处理结果

#### 分配策略

##### 随机分配策略

       轮询策略     调度服务器维护一个值,记录上次分配的IP,依次分配

                       但是轮询策略需要维护值,有额外的开销,而且这个值是互斥资源,需要加锁,降低了性能

         优缺点:

              优点:     实现容易,逻辑简单

              缺点:     在http重定向中,调度服务器只在用户第一次向网站发起请求时起作用,当调度服务器返回相应信息后,客户端的所有操作都是基于URL,此后浏览器不会与调度服务器产生关系

                   导致:

                        如果分配给用户的后端服务器发生故障,并且页面被缓存,当用户再次访问时,请求会发给故障的服务器,导致访问失败

                        这种方式无法真正意义的实现负载均衡,只是把请求平分给每台服务器

### DNS负载均衡

         DNS解析     数据包采用IP地址在网络中传播,为了方便记忆,使用域名访问网站     通过域名访问网站之前,先将域名解析成IP地址  这个过程由DNS完成,也就是域名服务器

         我们提交的请求不会直接发送给想要访问的网站,而是首先发给域名服务器,他帮我们解析成为IP地址并返回给我们,收到IP地址之后才会向IP发送请求,所以DNS有一个天然优势, 如果一个域名指向多个IP,每次域名解析时,将其中一个IP返回就可以实现负载均衡

#### 过程

        首先将域名指向多个后端服务器(也就是一个域名解析到多个IP上),再设置一下调度策略,剩下的负载均衡就可以交给DNS了

        当用户发起请求时,DNS会根据调度策略选择一个合适的IP返回给用户,用户向IP发起请求

#### 调度策略

随机     轮询     地域分配最近的服务器

        优缺点:

              配置简单    扩展性较强,可以解析多个IP

              不能定制调度策略     也无法真正了解每台服务器的负载情况,无法真正实现负载均衡     与http重定向一样 只是均分请求

         动态DNS可以解决后端故障,导致的DNS有缓存,IP仍能访问的问题

### 反向代理负载均衡

         反向代理服务器是位于实际服务器之前的服务器,所有的请求都经过反向代理服务器,服务器根据前后端的负载情况,将用户的请求交给后端服务器,再返回给用户

        优点:

               与http重定向相比,反向代理可以隐藏后端服务器,所有浏览器不予后端直接交互,可以保证调度者的控制权,提升集群的整体性能

               与DNS相比,可以更快的移除故障点

               可以真正意义上的负载均衡

        缺点:

               所有的请求都经过调度者,当请求量超过调度者的最大负载时,调度服务器的吞吐率降低会降低集群整体性能

        反向代理的一个问题:

              某台后端服务器处理了用户请求,并保存了session或者缓存,当用户再次发送请求时,无法保证仍由该后端服务器处理

                   解决:修改分配策略,以用户IP作为标识     在cookie中标注请求的id

     apache      ngnix

## Redis

Redis是一个开源的,遵守BSD协议,支持持久化(RDB,AOF)的内存存储的数据结构服务器,可用作 数据库,高速缓存,消息中间件,分布式锁

### 为什么使用redis

1. 为了提高效率,把热点数据存放在内存中而不是磁盘 (淘汰策略)
2. Redis与其他内存存储比较起来,有更多的数据类型
3. 支持主从,sub/pub,持久化

### 优化

1. 在redis中,key-value存在一个最小的内存大小,假设为512k,但是所存放的数据远远不到,可以把该数据存放在map中,然后把map作为value存放,达到扩容的效果
2. RDB+AOF
3. Master不做持久化,交给slave持久化

### 淘汰策略

带有生存时间的key称之为易失的🡪volatile

volatile-lru(least recently used) 从已经设置了过期时间的组中挑选最少使用的进行淘汰

vloatile-ttl 从已经设置了过期时间的组中挑选即将过期的进行淘汰

volatile-random 从已经设置了过期时间的组中随机挑选淘汰

allkeys-lru 对于所有的数据挑选最少使用的进行淘汰

allkeys-random 对于所有的数据随机淘汰

no envicition 不进行淘汰(默认使用)

### 命令&操作

#### 系统命令

打开客户端 redis-cli -h host -p port -a password

删除键 del key[key1…]

是否存在 exists key

序列化K,V dump key

反序列化 restore key

##### 生存时间

expire key seconds 设置生存时间,对已有的key在执行expire会刷新生存时间\

### 数据类型

#### string

#### list

#### set

#### hash

#### zset

#### hyperloglog

只存基数(也就是有几个数就存几,不存元素本身)

#### GEO

地理位置

### 持久化

#### RDB

RDB (redis database) 镜像全量持久化

##### 方式

1. redis调用fork()创建一个子进程
2. 子进程将数据集写到一个临时的rdb文件
3. 完成后将临时rdb替换旧文件

##### 优点

最大化redis性能:redis主进程会创建一个子进程进行持久化,主进程不会进行I/O,因此具有高性能

RDB是一个紧凑的文件,保存了redis在某一段时间的数据集,可以用来备份,在回复大数据集时,比aof快

##### 缺点

1. 数据完整性: 假设每五分钟进行一次持久化,假设redis故障,name会丢失这五分钟的数据
2. 每次保存RDB时,主进程都会fork一个子进程去执行保存操作,如果数据集比较庞大大,而且cpu十分紧张,会导致停止时间较长

#### AOF

append only file 增量持久化

##### 方式

1. 每次启动redis时都会读取aof文件,重置数据集
2. 每次有写操作的时候向aof尾追加命令

##### aof的重写方法

1. redis执行fork创建子进程
2. 子进程开始将新的aof写入临时文件,在写入期间,所有的新命令,父进程一边放入内存,一边追加进原aof文件末尾(这样可以保证即使停机,也可以保留信息在aof中)
3. 子进程重写完成后,向父进程发送一个信息,父进程接收后,将内存中的命令写入aof末尾

##### 优点

数据十分耐久,即使出现了故障,也会仅损失一秒数据(默认为一秒追加一次)

redis在aof文件过大时自动重写,这个过程是安全的

命令有序,易读

##### 缺点

体积大于rdb

根据fsyn策略,速度可能会比较慢 fync(设置每个多久追加一次)

#### RDB切换至AOF

1. 为最新的dump.rdb创建备份
2. 将备份移至安全区
3. 关闭or不关闭RDB,打开AOF

## Redis可能遇到的问题

### 缓存击穿/缓存穿透

指查询一个不存在的数据,缓存与数据库都不会命中

缓存击穿将导致不存在的数据每次请求都会去数据库请求获取,失去了缓存保护的意义

#### 解决方案

##### 缓存空对象

给空值做缓存,也就是在db与缓存中查询不到时,在缓存中存放该key,设置值为null,并设置一个较短的过期时间

### 缓存雪崩

由于缓存承载大量请求,有效的保护了存储层, 但是如果缓存由于某种原因不能提供服务,所有的请求都会到达存储层,将会导致存储层挂掉

#### 解决方法

##### 使用集群

##### 设置熔断器

### 热点key集中失效

当前的key是一个热点key,并发量十分大,在某一时刻数量较多的热点key过期,将会把大量的请求发送到数据库,导致数据库崩溃

##### 解决方案

为热点key设置不同的过期时间

## redis分布式锁

    setnx    如果存在则set    返回1    否则返回0

    getset   原子执行

    使用方式

        1.setnx(lockkey,当前时间+过期时间)    返回1,取锁成功    否则走2

        2.get(lockkey)    获取旧值oldTime,如果这个值比系统时间小,说明锁已超时,可以允许别的请求重新获取    走3

        3.计算newTime=当前时间+过期时间    然后getset(lockkey,newTime)    返回当前lockkey的值currentTime    如果currentTime与oldTime相同,说明getset成功,获取到了锁

        4.获取到锁之后,当先线程可以开始自己的业务处理,当处理完毕后,比较自己处理时间和对于锁设置的超时时间,如果小于锁设置的超时时间,则执行delete释放锁,如果大于超时时间,不需要对锁进行处理

## 消息队列

    原因:在高并发的环境下,来不及同步处理,请求往往会阻塞    使用消息队列,实现异步处理请求,减缓系统压力

    当不需要立刻获得结果,但是并发量有需要控制的时候,需要使用消息队列

    RabbitMQ    ActiveMQ    Kafka

    应用耦合    多应用通过消息队列对同一消息进行处理,避免接口失败导致过程失败

    异步处理    多应用对消息队列中同一消息进行处理,应用并发处理消息,比串行效率高

    限流削峰    避免流量过大导致挂掉

    模式

### 点对点模式

            消息队列    发送者   接收者

            消息发送者生产消息发送到队列,接收者从队列取出并消费    消费以后,队列中不保存

            特点:

                每个消息只有一个接收者(一旦被消费,不在消息队列中)

                发送者和消费者没有依赖性,发送者在发送后,不管接收者是否成功,不影响下次发送

                接收者在接收成功后需要向队列应答,以便删除队列中的信息

### 订阅发布

             角色主题    发布者    订阅者

            发布者将消息发送给topic,系统把消息传递给多个订阅者

            特点:

                每个消息可以有多个订阅者

                发布者与订阅者有时间的依赖性,针对某个topic,必须有第一个订阅者订阅,发布者才可以发布

                为了消费消息,订阅者需要提前订阅topic,并保持在线

    Kafka在于分布式架构，RabbitMQ基于AMQP协议来实现，RocketMQ/思路来源于kafka，改成了主从结构，在事务性可靠性方面做了优化。广泛来说，电商、金融等对事务性要求很高的，可以考虑RabbitMQ和RocketMQ，对性能要求高的可考虑Kafka。

# 微服务

## Spring框架中IOC和AOP

### IOC

控制反转，是一种设计模式。

一层含义是控制权的转移：由传统的在程序中控制依赖转移到由容器来控制；

第二层是[依赖注入](https://www.baidu.com/s?wd=%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3nhuBnym3rjuBm1m1mWDY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmzPHc1nWbd)：将相互依赖的对象分离，在spring配置文件中描述他们的依赖关系。他们的依赖关系只在使用的时候才建立。

### AOP

面向切面，是一种编程思想，OOP的延续。将系统中非核心的业务提取出来，进行单独处理。比如事务、日志和安全等。

Spring 的AOP和IOC都是为了解决系统代码耦合度过高的问题。使代码重用度高、易于维护。

不过AOP和IOC并不是spring中特有的，只是spring把他们应用的更灵活方便 。

## java 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

## spring cloud

         配置管理     服务发现     断路器     路由     微代理     事件总线     全局锁

1、Spring boot 是 Spring 的一套快速配置脚手架，可以基于spring boot 快速开发单个微服务；Spring Cloud是一个基于Spring Boot实现的云应用开发工具；

2、Spring boot专注于快速、方便集成的单个个体，Spring Cloud是关注全局的服务治理框架；

3、spring boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring boot来实现。

4、Spring boot可以离开Spring Cloud独立使用开发项目，但是Spring Cloud离不开Spring boot，属于依赖的关系。

## 微服务架构

是一种将单应用程序作为一套小型服务开发的方法，每种应用程序都在其自己的进程中运行，

并用轻量级机制（通常是HTTP资源的API）进行通信。这些服务是围绕业务功能构建的，

可以通过全自动部署机制进行独立部署。这些服务的集中化管理已经是最少的，它们可以用不同的编程语言编写，

并使用不同的数据存储技术。

## 注册与发现

在分布式系统中，各个子系统都是多个实例存在，这个时候必须要引入一个服务协调器，用于给调用方提供可用的调用提供者的命名消息。

服务协调器，如zookeeper，etcd，eureka

他们必须要有的特性：

- 本身高可用，由多个服务节点构成，就算有些节点挂掉也不影响正常运行，避免了单点故障。

- 本身是一个分布式，一致性的 k-v 存储系统。提供方启动的时候将自身配置信息向协调器中进行注册，提供方下线的时候向协调器进行反注册。服务调用方只要watch对应的配置，当协调器有对应的配置进行更改，就会通知调用方。

- 提供心跳机制，如果方程序意外挂掉，没有进行反注册，协调器也会超时剔除不可用的提供方。

- 服务协调器本身不提供负载均衡服务，负载均衡由调用方根据自己设定的策略进行路由调用。

传统的内置DNS虽然有些相像的功能，但是并不能用于服务注册与发现

* 本身不是高可用的，存在单点故障。
* 服务不能自动注册与自动发现。注册过程与反注册必须手动或者通过小程序通知DNS服务器。
* 服务调用方通过内部域名轮训DNS，DNS会自动根据负载均衡策略提供其中一个调用方的IP。这会存在两个问题。
  + 第一是DNS服务器本身支持的负载均衡策略很少而且无法根据自己的要求自己定制。
  + 第二是调用方第一次轮训DNS后会在本地缓存查询结果，在有效期内是直接针对某个调用方发起调用。调用提供者集群存在上线、下线操作，DNS是更新了，但是某个调用方还是会缓存已经下线的服务提供者导致调用失败。
* 没有提供心跳机制，提供方挂了都无法知晓。

# 面试

博客 cnblogs

## 问题?

### 问题1:

为什么 jdk8

现在他 不用头插

而是去 使用尾插

在网上 找不到啊